Evaluation of Fault-Tolerant Policies Using Simulation*

Anand Tikotekar [1], Geoffroy Vallée [1], Thomas Naughton [1], Stephen L. Scott [1], Chokchai Leangsuksun [2]

[1] *Oak Ridge National Laboratory*
Oak Ridge, TN 37831, USA
{tikotekaraa, valleegr, naughtont, scottsl}@ornl.gov

[2] *Louisiana Tech University*
Ruston, LA 71272
{box}@latech.edu

***Abstract*— Various mechanisms for fault-tolerance (FT) are used today in order to reduce the impact of failures on application execution. In the case of system failure, standard FT mechanisms are checkpoint/restart (for reactive FT) and migration (for pro-active FT). However, each of these mechanisms create an overhead on application execution, overhead that for instance becomes critical on large-scale systems where previous studies have shown that applications may spend more time checkpointing state than performing useful work.**

In order to decrease this overhead, researchers try to both optimize existing FT mechanisms and implement new FT policies. For instance, combining reactive and pro-active approaches in order to decrease the number of checkpoints that must be performed during the application's execution. However, currently no solutions exist which enable the evaluation of these FT approaches through simulation, instead experimentations must be done using real platforms. This increases complexity and limits experimentation into alternate solutions.

This paper presents a simulation framework that evaluates different FT mechanisms and policies. The framework uses system failure logs for the simulation with a default behavior based on logs taken from the ASCI White at Lawrence Livermore National Laboratory. We evaluate the accuracy of our simulator comparing simulated results with those taken from experiments done on a 32-node compute cluster. Therefore such a simulator can be used to develop new FT policies and/or to tune existing policies.

I. Introduction

High performance computing systems grow rapidly, with today's high-end systems moving from tera-scale to peta-scale class platforms. As these systems increase in size, the likelihood of component failure increases greatly, which ultimately leads to higher application level failures (typically parallel applications). applications).

A standard policy to address this issue is to periodically checkpoint applications and in the event of a failure restart the application. A more recent policy is to pro-actively avoid failures, moving parts of the application away from nodes for which a failure has been predicted. This latter approach allows one to avoid the impact of system failures on application execution. Unfortunately, it assumes failures are predictable which may not be the case, depending on applications and the execution platform characteristics. There is no one-size-fits-all solution. For instance, today's efficient checkpoint/restart schemes may be unsuitable for peta-flop machines with their massive node counts [1] [2]. Yet, it is possible that other solutions may be worse, thus making a particular checkpoint/restart scheme the only feasible solution for such a system. The point is that it is impossible to arrive at any conclusion without a proper evaluation and comparison with candidate solutions.

A new challenge for researchers is therefore to find appropriate fault tolerance policies for specific applications and execution platforms. This document presents a step in that direction, presenting a simulator for FT policies. This simulator allows one to evaluate the impact of various FT policies on application execution, based on characteristics of both the execution platform (e.g. number of nodes), FT mechanisms (e.g. their overhead), and applications (e.g. execution time).

This presented framework provides a tool for research into fault tolerance based on simulation, which removes the added complexity associated with experimentation incurred by work done using a real platform.

The remainder of this paper is organized as follows. We present the background of the proposed research work in the Section II including various FT policies that are generally used on real systems. In Section III, we describe our simulator, followed by an analysis in Section IV of different FT policies based on failure logs from Lawrence Livermore National Laboratory (LLNL). Section V shows the validation of the simulator. Lastly, Section VI presents our conclusion.

II. Background

A. Standard Fault Tolerance Policies

The main goal of a FT policy is to reduce the loss of an application's work time or reduce the execution overhead in the presence of failures. In this section we explain the various facets of a few standard FT policies, which were used to guide the implementation of our simulator.

We make the following assumptions in order to represent a particular FT policy: (i) An HPC system has an unknown MTBF (Mean Time Between Failures), i.e., all failures occur according to some unknown value for MTBF; (ii) an application completes after K failures.

* ORNL's work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

Let the lost work time or execution overhead be represented by L, and the application's original run time be equal to $n * mtbf$.

1) Exclusive Reactive Fault Tolerance: The central idea of the exclusive reactive fault tolerance policy is to periodically checkpoint applications. The main parameters of this policy are: (i) the checkpoint interval; (ii) the associated overhead for the application or checkpoint latency; (iii) the "time to checkpoint", i.e., the average time to complete one checkpoint; and (iv) rollback time due to application failure. Efficient checkpoint/restart schemes try to minimize the overhead for the application by parallelizing the checkpoint writing process. However, the longer it takes to complete a checkpoint, the higher the probability a failure will occur during this state saving period. This policy also depends on the availability of nodes to restart a failed application. Further, restart latency (in the case of synchronized checkpoint/restart) is another source of application overhead.

Let η be the associated overhead to the application for a single checkpoint and γ be the time to checkpoint.

$$z = mtbf - \lfloor \frac{mtbf}{interval + \gamma} \rfloor * (interval - \eta) \quad (1)$$

$$L = \infty \text{ if } interval > mtbf \quad (2)$$

$$L = \sum_{i=1}^{n} z \text{ for interval } < mtbf \quad (3)$$

The Equation 3 is only a partial result for the lost time. The Equation 3 characterizes lost time for n MTBFs. The lost time L can lead to more lost time if L is greater than $mtbf$. Therefore, L can be approximated to a multiple of MTBF, which can be then evaluated using Equation 3.

2) Exclusive Proactive Fault Tolerance: The basic tenet of the exclusive proactive fault tolerance policy is to predict impending system failures wherever possible and avoid application failures by moving running processes/threads away from the compute node that is about to fail. The policy has a high overhead to applications when failures are not predicted since applications restart from scratch (no checkpoint/restart mechanisms). The policy depends on failure prediction accuracy, migration overhead for the application, false alarm rate, and availability of spare nodes for migration.

The following example illustrates such a policy. Let the prediction accuracy be equal to 50% (i.e. 50% of the predicted failures are false prediction) and g(x) gives the number of failures that are correctly predicted.

$$L = \sum_{i=1}^{k-1} g(i) * mtbf * p \quad (4)$$

$$\text{where } 1 \leq g(x) \leq n-1 \text{ and where } 1 \leq x \leq k-1 \quad (5)$$

In this case, an application will only complete its execution if n consecutive failures with MTBF equal to $mtbf$ are predicted. The probability p that n consecutive failures are predicted correctly out of $2m$ failures that are subject to a 50% prediction accuracy can be calculated as follows:

$$p = \frac{\binom{m-n}{n} * (m-n+1)}{\binom{2m}{m}} \text{where} m \geq n \quad (6)$$

3) Reactive and Proactive Fault Tolerance: The fundamental premise behind the reactive and proactive fault tolerance policy is to acknowledge that all failures cannot be predicted, and therefore this policy combines (i) periodic checkpoint (reactive), and (ii) failure prediction with migration (proactive). For failures that are predicted, applications can be migrated thus saving the rollback time and restart latencies. For failures that cannot be predicted, applications are restarted from the last checkpoint thereby reducing re-computation by avoiding starting from scratch.

The execution overhead L in this case can be represented in much the same way as in the case of "exclusive reactive FT". The difference is that the MTBF gets stretched based on the failure prediction accuracy. Lets assume the failure prediction inaccuracy to be x (the accuracy is $1-x$)

$$mtbf' = \frac{1}{x} * mtbf \quad (7)$$

The Lost time or execution overhead L can then be represented using Equation 3.

4) Proactive Triggered Reactive Fault Tolerance: The proactive triggered reactive fault tolerance policy is used when applications are only checkpointed when failures are predicted. This is an alternative when migration overhead exceeds the checkpoint/restart overhead. In that case, the application loses $n*x/100*mtbf$ for the first n MTBFs at x percent prediction inaccuracy.

$$t = \frac{x}{100} \quad (8)$$

Therefore, the total lost time is a geometric series with rate as t.

$$L = mtbf * \sum_{i=1}^{\infty} n * t^i \quad (9)$$

$$L = mtbf * \frac{nt}{1-t} \quad (10)$$

B. Related Work

Our proposed research on simulation of FT policies has two basic components. So we present related work on both the components. Many simulation studies [3] [4] [5] evaluate the impact on metrics such as job turnaround time, average wait time, and average slowdown time, for various parallel job scheduling policies in the presence of system failures, but fault tolerant (FT) mechanisms/policies are not explored to find their impact on the same metrics. The authors in [6] propose a parallel simulator to compare various techniques for task allocation to multiple processors where processors are subject to failures. Although this study evaluates the metrics

such as job execution times, the objective of the authors is not to evaluate the impact of fault tolerant mechanisms. There have been efforts to study software dependability or failure behavior through simulation. Such efforts [7] [8] inject or simulate software or hardware failures that affect software to evaluate the impact of various fault-tolerant strategies. Oliner et al. in [9] study the performance implications of periodic checkpoints through simulation. Oliner et al. use LLNL's real system logs for their study. Our goal is to study and evaluate different FT policies and not just periodic checkpoint. Zhang et al. [10] [11] study performance metrics such as resource utilization, job slowdown, for scheduling strategies and not for fault-tolerant mechanisms.

The research on FT policies can be broken down into four policies as discussed in Section II-A. The research in reactive FT policies ranges from fault tolerant MPI [12], to fault tolerant schedulers [13], to projects like *déjà vu* [14] and numerous checkpoint/restart schemes [15], [16] The research in proactive FT policies can be broken down into failure prediction and migration. Hardware failure prediction is studied well in the literature [17], [18], [19] ranging from node failures to disk failures to IPMI logs. In [20], the authors explain how critical events can be predicted for pro-active management. In [20] Chakravorty and Kale explain how pro-active fault tolerance based on processor failure information can lead to migration of tasks from faulty processors to healthy nodes. Software rejuvenation along with checkpointing has been used in [21] to minimize the application execution time pro-actively. The FT policy of proactive combined with reactive policy is relatively less explored. But in [22] they describe how pro-active fault management can be used to decide whether to migrate the process or whether to checkpoint it.

III. Simulation Framework

Figure 1 shows the architecture of the simulator and how a particular simulation is performed. In our framework, the simulated system is defined by a set of *schemas*. For instance, different schemas define the application, the compute nodes, the FT policy and the failures (each individual schema are explained in detail in later sections). Doing so, it is possible to easily customize the simulation, modifying the different parameters of each schema. It is also possible to fine-tune the selected FT policy.

Each individual schema contains information about a specific element of the simulation. The simulator finds all the needed information via the schemas. As described in the next section, the configurable parameters could be the number of applications, number of nodes per application, total nodes, spare nodes, repair time, various overheads (such as checkpoint/migration/latencies), and various FT policies (such as proactive/reactive and failure logs).

A. Application Schema

Listing 1 describes the *application schema* as used by the simulator. Node count is the actual processor count of that particular application. We currently assume that only one application process/thread is executed on each node and that the node count of an application remains static throughout its lifecycle. For instance, in the case of proactive FT, we let the application fail if there is no spare node available for migration. One can easily extend the simulator to allow stacked processes by introducing a "stacked overhead" in the FT policies schema (please see Listing 3).

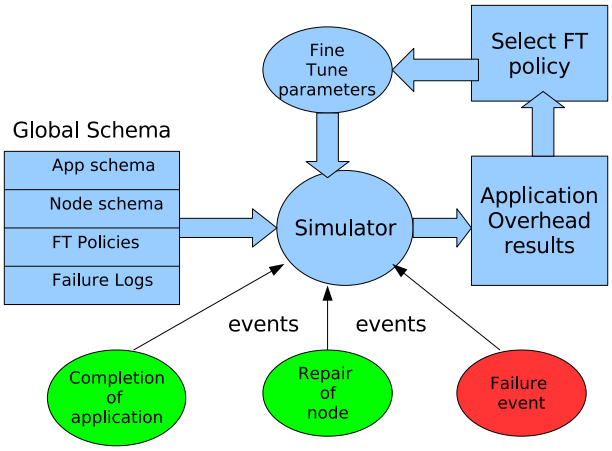

Fig. 1. Global Architecture of the Simulation Framework

Listing 1. Application schema as needed by the simulator

```
<App schema>
    <Application ID>...</Application ID>
    <Execution time>...</Execution time>
    <Node count>...</Node count>
</App schema>
```

B. Node Schema

The *node schema* as shown in Listing 2 stores information such as the number of spare nodes. The availability of spare nodes is critical for application's overall overhead as we will see in Section IV.

Listing 2. Node schema as needed by the simulator

```
<Node schema>
    <Total nodes>...</Total nodes>
    <Active nodes>...</Active nodes>
    <Spare nodes>...</Spare nodes>
</Node schema>
```

C. Fault Tolerance Policies

The *FT policies schema* shown in Listing 3 lists all supported FT policies by the simulator. The schema also describes the different overheads associated to each policy. For instance, the overheads in the schema could be "checkpoint overhead" or "migration overhead" or "Restart latency". The event element specifies the event on which a particular policy should be invoked. For instance, reactive policy should only be invoked when the failure event occurs. Similarly, the proactive policy must be invoked when failure is predicted.

Listing 3. FT policies schema as needed by the simulator

```
<FT policies schema>
    <policy>
        <Name>Reactive</Name>
        <event>...<event>
        <Overhead1>...</Overhead1>
        <Overhead2>...</Overhead2>
        <Overhead3>...</Overhead3>
    </policy>
    <policy>
        <Name>Proactive</Name>
        <event>...<event>
        <Overhead1>...</Overhead1>
        <Overhead2>...</Overhead2>
    </policy>
</FT policies schema>
```

D. Failure Logs

The *failure log schema* gives the details about each failure. Specifically, it contains information about failure event, i.e., the duration of the event (Mean Time To Recover - MTTR), the node affected, whether the failure was predictable. The predictability information is used by policies such as the proactive one.

Listing 4. Failure logs schema as needed by the simulator

```
<Failure logs schema>
    <Failure event>
        <start_time info>...</start_time info>
        <end_time info>...</start_time info>
    </Failure event>
    <node failure info>
        <node id>...</node id>
        <MTTR info>...</MTTR info>
    </node failure info>
    <predictable></predictable>
</Failure logs schema>
```

E. Simulator's Algorithm

The algorithm is based on the discrete event simulator principle and the implementation is Java based. The events in the simulator can be: (i) a failure event (based on failure logs), (ii) the completion of a running application, (iii) a node that comes back online after a failure. The simulation algorithm performs a series of steps when each of the above events occurs. The algorithm begins with a global schema as its input. The algorithm only receives a failure event at the start. If there is no failure event, events like "Repair event" and "completion of an application" are not triggered. However, a trace of the events is created in order to track event dependences when a failure is triggered by the simulator. The failure event comes from the failure logs schema, where the failure events are chronologically ordered.

Details about actions generated for each events are detailed in the following paragraphs.

TABLE I
INFORMATION FROM LLNL'S LOGS

Parameter	Description
Failure time	Time when failure occurred
Downtime (MTTR)	Downtime of a node
Node id	Failed node information

1) Failure Event: (i) extract information about the failed application from the application schema and the failure logs schema, (ii) check event dependences (such as "repair of nodes"), (iii) reorder events in chronological order, (iv) release jobs in queue and calculate overheads (such as loss of work time) based on past events, (v) execute the FT policy, (vi) update node lists for the failed applications.

2) Repair Event: (i) add node(s) to free/spare list.

3) Completion Event: (i) release the nodes held by job to free/spare list

4) Actions Based on FT Policy: (i) query schema for support, (ii) calculate overheads listed in the FT policies schema, for example calculate checkpoint overhead if the FT policy is reactive or calculate migration overhead if the policy is proactive.

IV. CASE STUDY USING LOGS FROM LLNL

In this section we evaluate standard FT policies listed in Section II-A against the failure logs from LLNL's ASCI white system. This case study describes the different set of results obtained from our simulator. Please note that even though the results are specific to the system in evaluation, we show how we can use this information to select a FT policy for a given set of parameters. This case study shows in more details how our simulator is working.

We simulate four space-shared parallel MPI jobs, each having 125 processes in a 512 node simulated cluster environment with 12 spare nodes. We use 1 month failure data starting from 7/15/2000 to 8/15/2000 from LLNL's ASCI White system logs. The default time for each simulated job is 744 hours. The system logs contain the following information as shown in Table I

The information in the LLNL logs do not contain failure prediction information. However, the failure prediction information is critical for evaluating proactive FT policies as described in Figure 3. Since this is our case study we experiment with different sets of prediction accuracies. For instance, 50 percent prediction accuracy means that about 21 failures are predicted in our 1 month failure data containing a total of 41 failures. Although the results differs based on which particular failures are predicted, it is of no particular importance to our discussion. This is due to the fact that we evaluate failure logs as they are, and it is fair to assume that a specific system may have such failure prediction distribution. We therefore randomly select number of predicted failures based on the required prediction accuracy during the given period of the LLNL logs.

TABLE II

VALUES FOR PARAMETERS FOR THE FIRST SET OF RESULTS

Parameter	Value
Number of applications running	4, each with 125 nodes
Total active nodes	500
Spare nodes	12
Time to Checkpoint	50 minutes /checkpoint
Checkpoint overhead to application	50 minutes/checkpoint
Migration overhead	1 minute/migration
False alarm rate	not set
MTTR	original values in logs

Next, we show the application failure distribution pertaining to LLNL logs (during the given period).

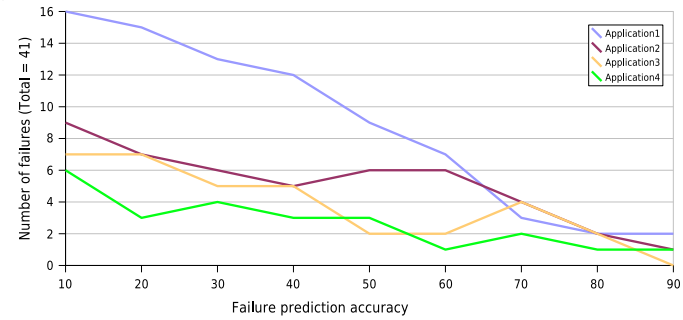

Fig. 2. Job Failure Statistics for 1 Month

A. First Set of Results

The results in Figure 3 to Figure 6 are based on the set parameters as shown in Table II

The parameters like checkpoint overhead and migration overhead are based on our knowledge of the existing overheads on our real platforms.

The set of results from Figure 3 to Figure 6 carry the same overhead for “Time to checkpoint” and “Checkpoint overhead to the application”. Efficient checkpoint/restart schemes allow checkpoint to work in parallel with the computation after a certain overhead (which is overhead to the application). In the next set of results we change the overheads and analyze its effects on application execution. The migration overhead is typically small compared to checkpoint overhead for large systems. This is due to the fact that only one process or one OS or one virtual machine needs to be migrated for a predicted failure (typically no synchronization between nodes is needed). We shall change this parameter also in the next set of results.

From Figure 3, it is clear that the availability of the four running applications depend on the failure prediction accuracy and also when the failure occurs in its lifetime. This is evident in the case of 60 percent and 70 percent predication accuracy. Although 70 percent prediction accuracy should mean better results, it may not necessarily be the case. The simple reason is that if an application fails toward the end, it loses more time since there is no checkpoint. For a researcher, information such as in Figure 3 is quite critical. After selecting the statistically significant sample of logs, the researcher is able to see the impact of the FT policy selected. The researcher can form an educated opinion as to what sort of prediction accuracy he/she must have in order to keep the execution overheads down. For instance, a conclusion from Figure 3 could be stated as follows: for longer running applications on our specific system, the prediction accuracy should be higher than 60 percent in order to keep the overhead below 50 percent for all applications.

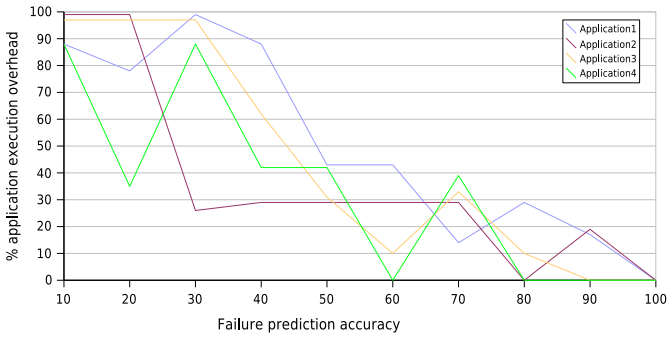

Fig. 3. Application Execution Overhead for Exclusive Proactive FT Policy (Based on a 1 Minute Migration Overhead)

Figure 4 shows a case where the FT policy is exclusively reactive. Application overhead in this case depends on the number of checkpoints and the overhead per checkpoint. Figure 4 shows that the best results are obtained when the checkpoint interval is not too low and not too high. When the checkpoint interval is too low (like 2 hours), most part of the overhead comes from the checkpoint overhead to the application. When the checkpoint interval is too high (like 256 hours), most part of the overhead comes from the computation rollback time. An example conclusion to be drawn from Figure 4 can be stated as follows: for our specific logs the interval between 8 hours to 64 hours is necessary in order to keep the application overhead below 20 percent.

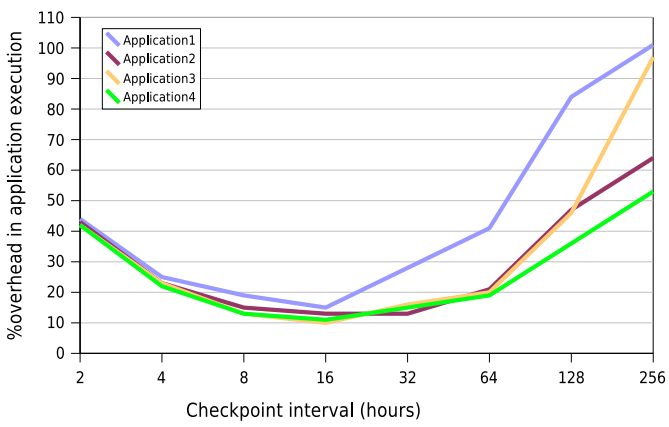

Fig. 4. Application Execution Overhead for Exclusive Reactive FT Policy (Based on a 50 Minutes Checkpoint Overhead

Figure 5 shows the results for a proactive combined with

reactive FT policy for application 1. This policy includes periodic checkpointing and failure prediction with migration. Figure 5 shows that the best results are obtained when the prediction accuracy is over 60 percent and checkpoint interval is between 16 and 32. A sample conclusion to be drawn from this can be stated as follows: For our specific logs, the prediction accuracy should be over 60 percent and checkpoint interval should be between 16 and 32 in order to keep the overhead below 20 percent. Now if we compare our conclusions for the three different FT policies, we can see that they are consistent. For instance, we would not like to keep checkpoint interval of 2 or 4 hours for 70 percent prediction accuracy because the overhead is more than that of exclusive proactive FT policy. Further, for the same 70 percent accuracy we also would not set very high checkpoint intervals because the advantage will be nullified in the form of rollback time.

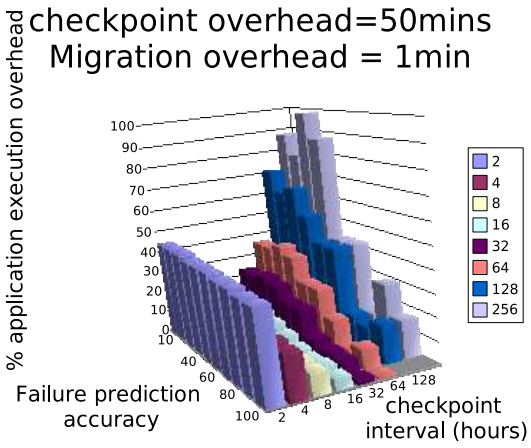

Fig. 5. execution overhead for Proactive and reactive FT policy for Job 1

Figure 6 describes the results for application 2. The results are similar to Figure 5. The results for application 3 and 4 are not shown for the sake of brevity.

Figure 7 shows yet another FT policy. This policy as described earlier uses checkpoints as triggered by failure prediction. The downside of this policy is the checkpoint overhead but the plus side is that rollback time is limited. Comparing Figure 7 with Figure 3, we can see that the impact is approximately the same, although it may not be necessarily the same. In these two case, while the checkpoint overhead (at 50 minutes/checkpoint) is much larger than migration overhead (1 minute/migration), the rollback time is much lesser in pro triggered FT policy than Fully proactive FT policy.

It is now clear that without the simulator a researcher having access to a failure prediction mechanism and checkpoint/restart scheme would need a lot more time and effort just to try out various FT policies to ensure the minimum overhead. Moreover, he/she may have to settle for a sub-optimal FT policy without some kind of a simulator.

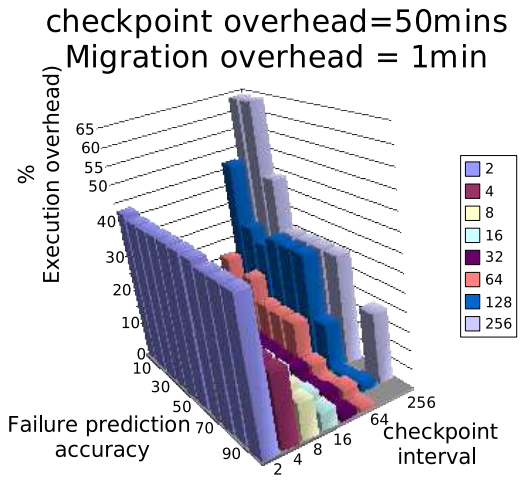

Fig. 6. Execution Overhead for Proactive and Reactive FT Policy for Job 2

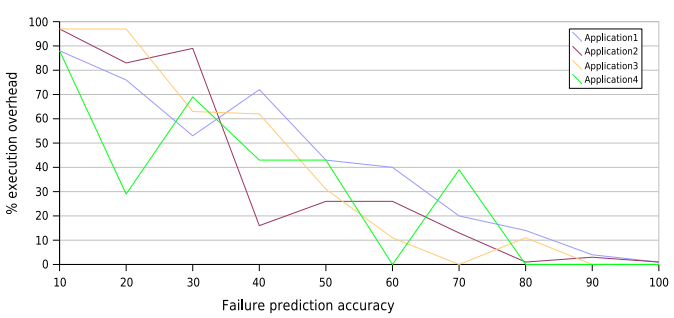

Fig. 7. Execution Overhead for Proactive Trigger Reactive FT (Based on a 50 minutes Checkpoint Overhead)

B. Second Set of Results

In this section we set different values for our parameters than for the first set of results. The new values for the parameters appear in Table III. Specifically, we decreased the checkpoint overheads 5 times and increased the migration overhead 50 times.

The impact of increasing the migration overhead even by 50 times is not significant when we compare Figure 8 with Figure 3. This implies that migration takes place less frequently depending on the failure prediction accuracy. The conclusion from this result is that with less frequent failures,

TABLE III
VALUES FOR PARAMETERS FOR THE SECOND SET OF RESULTS

Parameter	Value
No of applications running	4 each with 125 nodes
Total active nodes	500
Spare nodes	12
Time to Checkpoint	10 minutes /checkpoint
checkpoint overhead to application	10 minutes/checkpoint
Migration overhead	50 minute/migration
False alarm rate	not set
MTTR	original values in logs

migration overhead is an insignificant quantity compared to the rollback time lost. Approximately all the overhead comes from the rollback time.

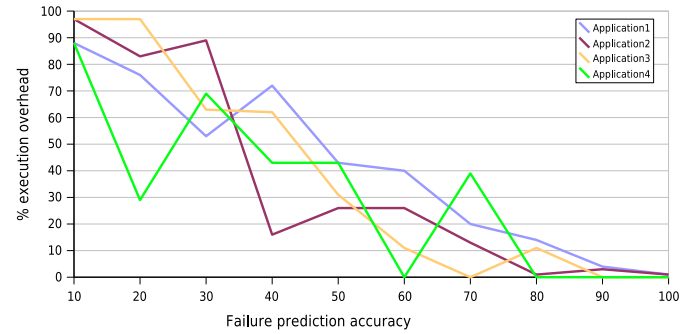

Fig. 8. Application execution overhead for exclusive proactive FT policy (Based on a 50 Minutes Migration Overhead)

Comparing Figure 9 with Figure 4, we can see the impact of the checkpoint overhead. In this set of results, checkpoint overhead was reduced to 10 minutes from 50 minutes in the first set of results. The reduction in execution overhead is strong for interval between 2 and 16. For instance, the execution overhead for job 1 in Figure 4 for 2 hours interval is 4 times more than that of Figure 9 with the same interval. This is expected since low interval means more number of checkpoints and therefore more checkpoint overhead to the application. The conclusion drawn from this result is that checkpoint overhead is a significant quantity when the interval is low.

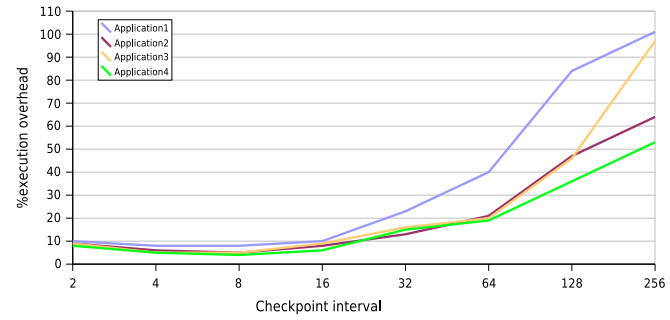

Fig. 9. Application Execution Overhead for Exclusive Reactive FT Policy (Based on a 10 Minutes Checkpoint Overhead)

We now show the impact of the MTTR (Mean time to repair) on application's execution overhead. Until now, we have been using original MTTR values from the LLNL logs. we now describe the impact of modified MTTR on the applications. Table IV shows the static parameters for this operation. Specifically, we set 32 hours for our checkpoint interval and 50 percent as our prediction accuracy.

Figure 10 shows a scenario where there are no spare nodes. As can be seen from the figure, as the MTTR increases the execution overheads generally increase. The reason for drop of execution overhead in some cases is not surprising since

TABLE IV
VALUES FOR PARAMETERS FOR THE SECOND SET OF RESULTS

Parameter	Value
Time to Checkpoint	10 minutes /checkpoint
checkpoint overhead to application	10 minutes/checkpoint
Migration overhead	50 minute/migration
checkpoint interval	32 hours
Failure prediction accuracy	50 percent

it depends on the timing of the "Repair event" of the node. For example, the reduced MTTR values disrupt the original chronological order of the "Repair event". This causes node lists of applications to change and therefore their failure pattern changes. The drop in one application's execution overhead should be offset by increase in the other. But the trend overall is that of increasing execution overhead for increasing MTTR values. (Please note that while only MTTR parameter is varied, the results show the total impact, for instance these results also contain checkpoint, migration, rollback overhead)

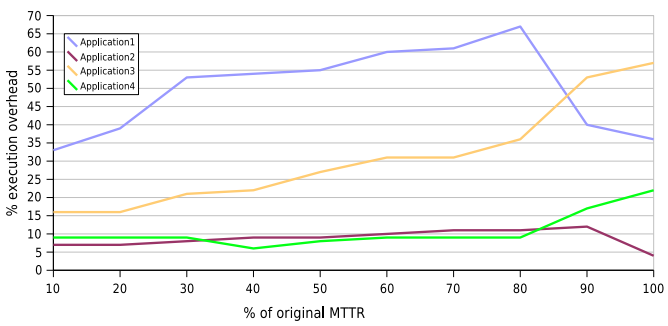

Fig. 10. Application Execution Overhead for Exclusive Reactive FT Policy (No Spare Nodes)

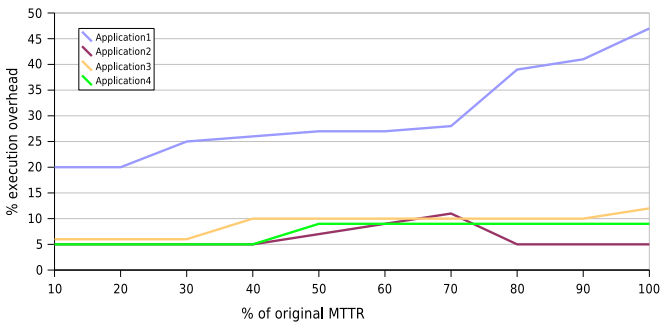

Fig. 11. Application Execution Overhead for Exclusive Reactive FT Policy With 4 Spare Nodes

Figure 11 shows the impact of MTTR reduction when the number of spare nodes is increased to 4. The increase in the number of spare nodes reduces the execution overhead since applications have to wait less for the "Repair event".

To summarize, we performed the case study based on LLNL's ASCI white system using our simulator. We described two sets of results. The first set of results evaluated different

TABLE V

PARAMETERS FOR REAL PLATFORM AND SIMULATION

Parameter	Value
Node count	16
Time to Checkpoint	90 seconds /checkpoint
checkpoint overhead to application	70 seconds/checkpoint
Restart latency	1 minute/restart
Migration overhead	1 minute/migration
Failure prediction accuracy	simulated

standard policies against the set of parameters shown in Table II. The best results are obtained when we used "Proactive combined with reactive FT policy". The second set of results were meant for tuning the various parameters and evaluated exclusive proactive and exclusive reactive FT policies against the set of parameters shown in Table III. Further, the second set of results also evaluated a "proactive combined with reactive FT policy" with reduced values of MTTR and with different number of spare nodes. This completes our case study for the simulator. In the next section we present the validation of our simulator.

V. Validation of the Simulator

We evaluated "exclusive reactive FT", "proactive combined with reactive FT" policies on real platform. Our platform consists of a 40 nodes cluster, with each node having 768MB of memory. For our reactive FT policy, we use BLCR checkpoint module. For our proactive FT policy we use virtual machines, specifically Xen [23], and their live migration [24] capability. For our real platform, we do not have failure logs or failure statistics, and therefore we simulate failures to evaluate the impact of FT policies.

We executed different configurations of the HPCC benchmark application suite. The 16 and 32 node experiments used HPCC problem sizes of 9,000 and 10,700 respectively. Table V shows various parameters for our platform of 16 nodes.

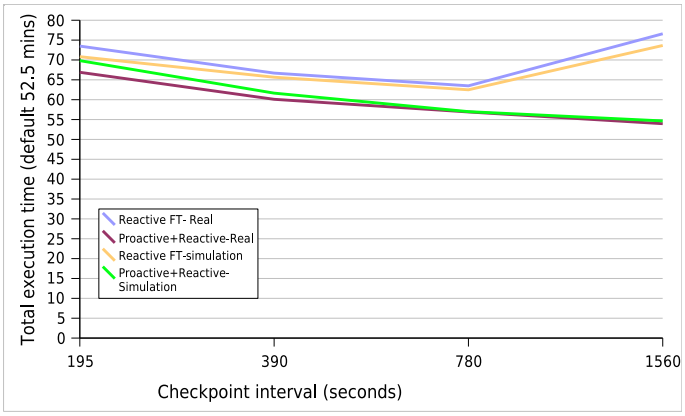

Fig. 12. Real vs Simulation for 16 Nodes (Failure at 20 Minutes)

Figure 12 shows the comparison between real and simulated results for 16 nodes. The error between the real and simulated results does not exceed 4 percent for both policies.

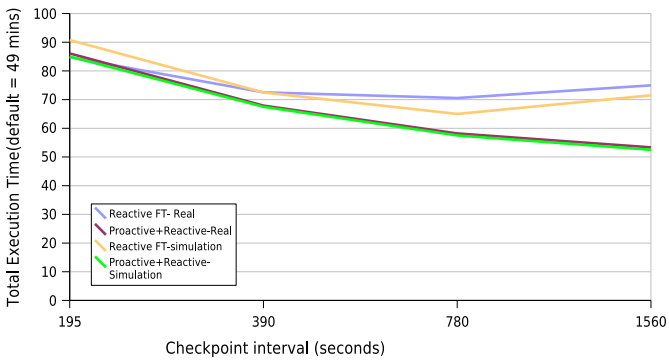

Fig. 13. Real vs Simulation for 32 nodes (Failure at 20 Minutes)

TABLE VI

PARAMETERS FOR REAL PLATFORM AND SIMULATION

Parameter	Value
Node count	32
Time to Checkpoint	150 seconds /checkpoint
checkpoint overhead to application	150 seconds/checkpoint
Restart latency	4 minute/restart
Migration overhead	1 minute/migration
Failure prediction accuracy	simulated

The checkpoint and migration overheads are the average overheads for BLCR and Live migration respectively on our platform for 32 nodes as shown in Table VI. Figure 13 shows the comparison between real and simulated results for 32 nodes. Since the parameters for simulation have average values for checkpoint overheads, restart latencies and migration overheads, the results can not entirely match that of the real ones. For instance, we took restart latency as 4 minutes in our simulation. But it varied back and forth between different restarts (from 1 to 8 minutes). The error between real and simulated in this case does not exceed 8 percent.

VI. Conclusion

This document presents a simulator framework for the evaluation of fault tolerance policies. We argued that there is no one-size-fits-all FT policy, and the challenge lies in finding/evaluating a policy that provides the best results for the application and execution platform. To that end, we presented our simulation architecture and algorithm. We presented a case study based on the evaluation of standard FT policies using failure logs taken from LLNL's ASCI White system. Furthermore, we illustrated how the simulator can be used to draw conclusions about appropriate FT policies for an application with a given dependability threshold. For instance, our case study results indicated that the most appropriate FT policy for the ASCI White system is a hybrid approach, "proactive combined with reactive".

The FT simulator complements existing work, such as FAIL-MPI [8], by providing a tool for off-line studies of FT policies. This eliminates the overhead and added complexity associated with investigations that rely on actual execution for

FT policy evaluation.

In order to evaluate our simulator we compared results from the simulator with experimental results from a 32-node compute cluster. This comparison showed that for this platform, results from the simulator are comparable to results from experimentation.

The current framework is based on a default reliability profile taken from failure logs from LLNL's ASCI White. However, this default profile can be replaced with other failure rates by using alternate logs. This capability enables the study of FT policies on different virtual platforms, with different availability characteristics. This capability is interesting especially since the different execution platforms have different characteristics based on their hardware, their scale, and their available fault tolerance mechanisms (e.g. process migration or checkpoint/restart). For computing centers that maintain failure logs, it is also possible to provide the platform's characteristics to researchers that can then run simulations with different FT policies during the exploration phase of the research project without having to occupy the real hardware.

The current framework has been validated by comparing simulation output with experimental results taken from both 16-node and 32-node clusters. In the worst case, the results from the simulator differ to experimental results by 8 percent. However, results at that scale cannot be assumed as representative for large-scale systems. On the other hand, it is difficult to access availability statistics for large scale systems since such logs are often considered to be sensitive data. However, we plan to continue the validation effort using different platforms for which we are creating such logs. We also plan to use the simulator for the evaluation of new FT strategies.

REFERENCES

[1] K. Yelick, "The software challenges of petascale computing," *HPCwire interview*, 2006.

[2] R. Oldfield, "Investigating lightweight storage and overlay networks for fault tolerance," in *HAPCW'06: High Availability and Performance Computing Workshop*. Santa Fe, New Mexico, USA: Held in conjunction with LACSI 2006, OCT 2006.

[3] H. D. Karatza, "Gang scheduling performance on a cluster of non-dedicated workstations," in *SS '02: Proceedings of the 35th Annual Simulation Symposium*. Washington, DC, USA: IEEE Computer Society, 2002, p. 235.

[4] H. Franke, J. Jann, J. E. Moreira, P. Pattnaik, and M. A. Jette, "An evaluation of parallel job scheduling for asci blue-pacific," in *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM Press, 1999, p. 45.

[5] H. Rajaei, M. Dadfar, and P. Joshi, "Simulation of job scheduling for small scale clusters," in *WSC '06: Proceedings of the 38th conference on Winter simulation*. Winter Simulation Conference, 2006, pp. 1195–1201.

[6] J. A. Cemisid, "A parallel simulator for task allocation in a distributed system subject to breakdowns," 2002. [Online]. Available: citeseer.ist.psu.edu/506743.html

[7] R. Goswami, K.K.; Iyer, "Simulation of software behavior under hardware faults," in *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third Internat Symposium on, Vol., Iss., 22-24 Jun 1993*, 1993, pp. 218–227.

[8] T. H. William Hoarau, Pierre Lemarinier, "Fail-mpi: How fault-tolerant is fault-tolerant mpi," 2006. [Online]. Available: http://www.lri.fr/ lemarini/papers/cluster06.pdf

[9] A. J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta, "Performance implications of periodic checkpointing on large-scale cluster systems," in *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*. Washington, DC, USA: IEEE Computer Society, 2005, p. 299.2.

[10] Y. Zhang, A. Sivasubramaniam, J. Moreira, and H. Franke, "A simulation-based study of scheduling mechanisms for a dynamic cluster environment," in *ICS '00: Proceedings of the 14th international conference on Supercomputing*. New York, NY, USA: ACM Press, 2000, pp. 100–109.

[11] S. Gokhale, M. Lyu, and K. Trivedi, "Reliability simulation of fault-tolerant software and systems," 1997. [Online]. Available: citeseer.ist.psu.edu/gokhale97reliability.html

[12] G. E. Fagg, A. Bukovsky, and J. J. Dongarra., "Harness and fault tolerant mpi." *Parallel Computing*, 2001.

[13] "Slurm: Simple linux utility for resource management," http://www.llnl.gov/linux/slurm/slurm.html. [Online]. Available: http://www.llnl.gov/linux/slurm/slurm.html

[14] "Deja vu software," http://www.californiadigital.com/sw.html. [Online]. Available: http://www.californiadigital.com/sw.html

[15] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," *Proceedings of USENIX Winter1995 Technical Conference, New Orleans USA*, pp. 213–224, Jan. 1995. [Online]. Available: citeseer.ist.psu.edu/plank95libckpt.html

[16] G. Stellner, "Cocheck: Checkpointing and process migration for mpi." *In Proceedings of the 10th International Parallel Processing Symposium (IPPS '96), Honolulu*, 1996.

[17] P. Apparao and G. Averill, "Firmware based platform reliability," *Intel magazine*, 2005.

[18] G. Hamerly and C. Elkan, "Bayesian approaches to failure prediction for disk drives," *In Proceedings of the eighteenth international conference on machine learning*, 2001.

[19] N. Talagala and D. Patterson, "An analysis of error behavior in a large storage system," *Technical Report UCB/CSD-99-1042, University of California, Berkeley, Computer Science Division*, 1999.

[20] S. Chakravorty, C. Mendes, and L. Kale, "Proactive fault tolerance in large systems." *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11). IEEE Computer Society*, 2005.

[21] S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi, "Minimizing completion time of a program by checkpointing and rejuvenation," *Proceedings of the 1996 ACM SIGMETRICS Conference*, 1996.

[22] Y. Li and Z. Lan, "Exploit failure prediction for adaptive fault-tolerance in cluster computing," *CCGrid*, vol. 0, pp. 531–538, 2006.

[23] E. Dragovic, P. Barham, K. Fraser, S. Hand, T. H. A. Ho, R. Neugebauery, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *In the Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[24] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*. Boston, MA: USENIX, May 2-4, 2005.